Chapter 2
Basic Supervisory Control Problem

Discrete event systems are systems that are characterized by the sequences of events that they can accept or execute. From an abstract point of view it is not important whether the system actually generates events or whether it restricts the possible events that are generated somewhere else. The important point is that the behavior of the system can be described by sequences of events.

Control of discrete event systems was first introduced by P. J. G. Ramadge and W. M. Wonham. See [45, 52, 59] for an overview on the subject. In the framework proposed by Ramadge and Wonham, a system is described by the sequences of events that it can generate, i.e. the language. Also the sequences that represent completed tasks are taken into account. This framework can handle deterministic systems only. Another drawback is that systems are considered on their own. But, as was discussed in the introduction, systems have to be considered in combination with their environment. It will be shown that a framework based on complete sequences is not suited for considering systems inside an environment. In this thesis a framework will be presented that guarantees the correct behavior of implementations in any environment. The framework is based on failure semantics [10, 11, 23]. Failure semantics provides a theoretical foundation to reason about the behavior of nondeterministic discrete event systems.

A discussion on the motivation of this framework will be given in Section 2.4. It will be shown that the supervisory control framework based on failure semantics is a flexible and elegant method. It guarantees deadlock free

behavior under all circumstances, it allows for powerful specifications, it forms a sound basis for modular control, and it can handle nondeterminism without extra effort.

2.1. Languages and Automata

In the literature several models are proposed to describe the behavior of discrete event systems. To name a few: languages, automata, transition systems, Petri nets, boolean expressions, process algebras, etc. In this thesis we will use representations based on languages, automata, and failure semantics. Failure semantics will be discussed in the next section. In this section formal definitions concerning languages and automata are given. Also some well known results from the literature are recalled. See [26] for a more detailed introduction to languages and automata.

Definition 2.1 Let Σ denote the finite set of all possible events or event labels. A set of events is usually called an *alphabet*. A *trace* or *string* is a finite sequence of events, $\sigma_1\sigma_2\ldots\sigma_n$ with $\sigma_i \in \Sigma$ for all $i \in 1\ldots n$. The *length* of a trace is the number of events in the trace. Let ε be the empty trace, i.e. the sequence of events with length 0. Note that $\varepsilon \notin \Sigma$. Σ^n denotes the set of traces with length n. Let $\Sigma^* = \bigcup_{i=0}^{\infty} \Sigma^n$. It denotes the set of all finite traces with events in Σ. Let $\Sigma^+ = \bigcup_{i=1}^{\infty} \Sigma^n = \Sigma^* - \{\varepsilon\}$. A *language* is a set of traces, i.e. a subset of Σ^*. The language of discrete event system A is denoted by $\mathrm{L}(A)$.

An *automaton* is a representation of a discrete event system based on states and transitions between states. In the literature also the terms labeled transition system, finite state machine, and generator are used. The differences between these models are relatively small. In this thesis we will use the term automaton to denote all these forms. When necessary we will state explicitly which form is used.

Definition 2.2 An *automaton* in its basic form is a four tuple[1]

$$X \quad = \quad (\Sigma(X),\, Q(X),\, \delta(X),\, Q_0(X)),$$

where

$\Sigma(X)$	$\subseteq$	Σ	is the set of events,
$Q(X)$			is the set of states,
$\delta(X)$	:	$Q(X) \times \Sigma(X) \to 2^{Q(X)}$	is the transition function,
$Q_0(X)$	$\subseteq$	$Q(X)$	is the set of initial states.

[1] In the classical automata theory [26] a fifth element describing the marked states is included. These states are needed to represent completed traces. As completed traces are not used in this thesis this fifth element is not included in the basic form of an automaton.

When needed, extra elements will be added to this basic form. In the sequel the notation $\delta(X, q, \sigma)$ will be used instead of $\delta(X)(q, \sigma)$.

Figure 2.1 shows a graphical representation of an automaton. The nodes of the graph correspond with the states of the automaton. Let q, $q' \in Q(X)$ and $\sigma \in \Sigma(X)$. Then $q' \in \delta(X, q, \sigma)$ if and only if there exists an arrow from node q to node q' labeled σ. Small arrows that do not start at a node point to the initial states.

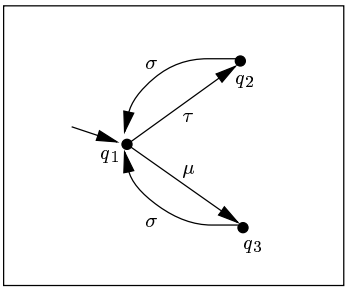

Figure 2.1: Graphical representation of an automaton.

Automata can be seen as machines that generate sequences of events. Initially an automaton is in one of its initial states. From state q automaton X can make a transition to state q' while generating event σ if $q' \in \delta(X, q, \sigma)$. If $\delta(X, q, \sigma) = \emptyset$ then X cannot make a transition from state q labeled σ. The set of traces that automaton X can generate in this way is called the *language of automaton* X, and denoted $\mathrm{L}(X)$. The term 'language of an automaton X' can be regarded as short for 'the language of a discrete event system represented by automaton X'.

The transition function can be extended naturally to traces, languages, and sets of states. Let $q \in Q(X)$, $\mathcal{Q} \subseteq Q(X)$, $\sigma \in \Sigma(X)$, $s \in \Sigma(X)^*$, and $K \subseteq \Sigma(X)^*$. Define

$$\begin{aligned}
\delta(X, \mathcal{Q}, \varepsilon) &= \mathcal{Q}, \\
\delta(X, \mathcal{Q}, \sigma) &= \bigcup_{q' \in \mathcal{Q}} \delta(X, q', \sigma), \\
\delta(X, \mathcal{Q}, s\sigma) &= \delta(X, \delta(X, \mathcal{Q}, s), \sigma), \\
\delta(X, \mathcal{Q}, K) &= \bigcup_{s \in K} \delta(X, \mathcal{Q}, s).
\end{aligned}$$

In the sequel we will deliberately confuse an element with the singleton set containing the element, whenever the meaning is clear from the context. For example, if the result of the transition function is a singleton set then we will write $q = \delta(X, \mathcal{Q}, s)$ instead of $\{q\} = \delta(X, \mathcal{Q}, s)$. Similarly define $\delta(X, q, s) = \delta(X, \{q\}, s)$ and $\delta(X, q, K) = \delta(X, \{q\}, K)$.

For notational convenience we will often omit the initial states as argument of the transition function.

$$\delta(X, s) = \delta(X, Q_0(X), s).$$

The language of automaton X is formally defined by

$$\mathrm{L}(X) \quad = \quad \{s \in \Sigma(X)^* : \delta(X, s) \neq \emptyset\}.$$

The definition of automaton given above is sometimes in the literature referred to as *nondeterministic automaton*. These automata are called nondeterministic because from the observation of a generated trace it cannot be uniquely determined which state the automaton has reached after execution of this trace. An automaton, X, is called *deterministic* if the initial state set is a singleton set, and if at each state $q \in Q(X)$, and for each event $\sigma \in \Sigma(X)$, $\delta(X, q, \sigma)$ contains at most one element. After observation of a trace $s \in \mathrm{L}(X)$, with X a deterministic automaton, it is always clear which state X has reached. Namely, the only element of the singleton set $\delta(X, s)$. (So, we will write $q = \delta(X, s)$ instead of $\{q\} = \delta(X, s)$.)

In the rest of this section we will discuss some properties of traces, languages and automata.

Definition 2.3 Let $s, t \in \Sigma^*$ such that $s = \sigma_1\sigma_2 \ldots \sigma_n$ and $t = \tau_1\tau_2 \ldots \tau_m$. The *concatenation* of s and t is the trace

$$st \quad = \quad \sigma_1\sigma_2 \ldots \sigma_n\tau_1\tau_2 \ldots \tau_m.$$

The concatenation of two languages $K,\ L \subseteq \Sigma^*$ is the language

$$KL \quad = \quad \{st : s \in K \ \wedge \ t \in L\}.$$

The *prefixes* of trace $s \in \Sigma^*$, denoted $\overline{s}$, are all traces that can be extended to s.

$$\overline{s} \quad = \quad \{v \in \Sigma^* : \exists t \in \Sigma^* \text{ s.t. } s = vt\}.$$

The *prefix closure* of a language $K \subseteq \Sigma^*$ is the set of all prefixes of traces in K.

$$\overline{K} \quad = \quad \bigcup_{s \in K} \overline{s}.$$

A language is called *prefix closed* if it is equal to its prefix closure, i.e. $K = \overline{K}$. The *choice* between languages $K,\ L \subseteq \Sigma^*$ is the language

$$K + L \quad = \quad K \cup L.$$

The *repetitive closure* of language $K \subseteq \Sigma^*$ is the language

$$K^* \quad = \quad \bigcup_{n\in\mathbb{N}}\{s_1 \ldots s_n : s_1, \ldots, s_n \in K\}.$$

In the sequel we will deliberately confuse a trace $s \in \Sigma^*$ with the language $\{s\}$. So an expression of the from $\sigma(\tau + \mu)^*$ denotes the language consisting of all traces of the form

$$\sigma,\ \sigma\tau,\ \sigma\mu,\ \sigma\tau\tau,\ \sigma\tau\mu,\ \sigma\mu\tau,\ \ldots$$

These kind of expressions are called *regular expressions* [26].

The *next event function* λ gives all events that are possible after a string.

$$\lambda(K, s) \quad = \quad \{\sigma \in \Sigma : s\sigma \in K\}.$$

The *ρ-function* is the complement of the next event function. It gives all events that cannot be executed after a string.

$$\begin{aligned} \rho(K, s) \quad &= \quad \Sigma - \lambda(K, s) \\ &= \quad \{\sigma \in \Sigma : s\sigma \notin K\}. \end{aligned}$$

The *language after trace* $s \in K$ is defined to be

$$K/s \quad = \quad \{v \in \Sigma^* : sv \in K\}.$$

Two traces s, $s' \in K$, are called *Nerode equivalent* if the languages after s and s' are equal.

$$s \equiv_K s' \quad \Longleftrightarrow \quad K/s = K/s'.$$

Let $[s]_K$ denote the equivalence set induced by the Nerode equivalence relation, containing trace s.

$$[s]_K \quad = \quad \{s' \in K : K/s = K/s'\}.$$

If an automaton can generate trace $s \in \Sigma^*$ then it is clear that it can also generate all its prefixes. So, a language generated by an automaton is always prefix closed. Given any nonempty prefix closed language, there exists an automaton that generates this language. However, the automaton may have an infinite state space. Let the *canonical automaton representation* of the nonempty prefix closed language K be the deterministic automaton defined by

$$(\Sigma,\ Q(K),\ \delta(K),\ Q_0(K)),$$

where

$$\begin{aligned} Q(K) \quad &= \quad \{[s]_K : s \in K\}, \\ Q_0(K) \quad &= \quad [\varepsilon]_K, \end{aligned}$$

and for all $[s]_K \in Q(K)$

$$\delta(K,[s]_K,\sigma) \quad = \quad \begin{cases} [s\sigma]_K & \text{if } \sigma \in \lambda(K,s), \\ \emptyset, & \text{otherwise.} \end{cases}$$

Language K is called *regular* if $Q(K)$ is finite.

For any language $K \subseteq \Sigma^*$ there exists a deterministic automaton that generates K. So also for all languages generated by nondeterministic automata. In other words, for any nondeterministic automaton there exists a deterministic automaton that generates the same language. Let X be a (nondeterministic) automaton. Define

$$\mathrm{Det}(X) \quad = \quad (\Sigma(X),\ 2^{Q(X)},\ \delta(\mathrm{Det}(X)),\ \{Q_0(X)\}),$$

where

$$\delta(\mathrm{Det}(X),\mathcal{Q},\sigma) \quad = \quad \{\delta(X,\mathcal{Q},\sigma)\} \qquad \text{for all } \mathcal{Q} \in 2^{Q(X)} \text{ and } \sigma \in \Sigma(X).$$

The state space of $\mathrm{Det}(X)$ is the power set of $Q(X)$. It contains as states the subsets of $Q(X)$. The size of $Q(\mathrm{Det}(X))$ is $2^{|Q(X)|}$, i.e. exponentially in the size of the state space of Q. Worst case, every subset of $Q(X)$ is needed to represent $\mathrm{Det}(X)$. Therefore, the conversion is said to have a worst case exponential complexity. However, in practice only those subsets are needed that are reachable by a trace of $\mathrm{L}(X)$. So this complexity is only worst case [26].

As $\mathrm{Det}(X)$ is deterministic the result of $\delta(\mathrm{Det}(X),\mathcal{Q},\sigma)$ is a singleton set. It contains as single element the set $\delta(X,\mathcal{Q},\sigma)$. Likewise, the set of initial states contains one element: $Q_0(X)$.

2.2. Processes

Traditionally in discrete event control only deterministic systems are considered. Two deterministic systems are considered equivalent if they generate the same language. Most results are stated in terms of languages. Automata are only used to show how results can be computed. This has the advantage that the results are not dependent on the automaton representation but also hold for other representations such as Petri nets and process algebras. In this thesis we want to extend the results to nondeterministic discrete event systems. It will be shown that the language alone is not sufficient to describe the behavior of a nondeterministic discrete event system.

Example 2.4 Consider a vending machine that hands out a cookie or a chocolate bar in exchange for a coin. In Figure 2.2 the representations of three vending machines are given by finite state automata. All three machines can generate the same language but will behave differently. Therefore, it is not sufficient to describe their behavior by the language that they can generate. How the machines behave is best illustrated by letting a user operate the machines.

After a client inserts a coin, the first machine will always hand out what the user requests. It will never refuse to give a cookie nor a chocolate bar. If a customer insists on having a chocolate bar from the second machine and this machine is in the state in which it can only hand out a cookie, then no event can be executed and the system is said to be in *deadlock*. The machine will however never refuse to hand out a cookie. The third machine can sometimes refuse to give a cookie and sometimes refuse to give a chocolate bar. But it cannot refuse both at the same time. If a user requests either of the sweets, no matter if it is a cookie or chocolate bar, then the machine cannot refuse and it must hand out one of them.

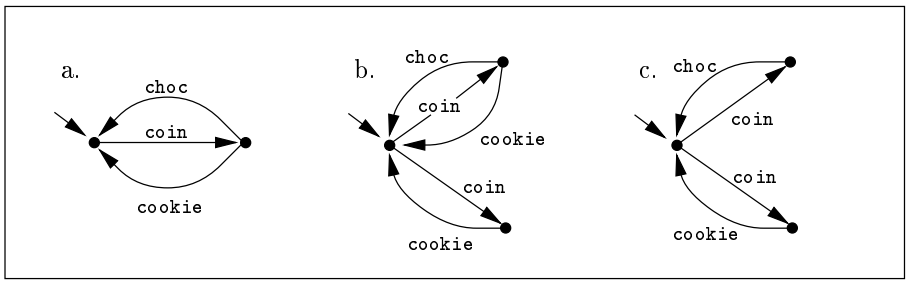

Figure 2.2: Models of a vending machine.

To describe the behavior of the machines it is necessary to not only describe the events that can be executed, i.e. the language, but also the event sets that can be refused. This is the basis of failure semantics [23]. A machine can refuse event set $R \subseteq \Sigma$ after string s, if it can reach a state by executing string s, and it cannot execute any event of event set R in this state. The event sets that can be refused are called *refusals*. A set of refusals is called a *refusal set*. Let $\mathrm{ref}(X, s)$ denote the refusal set of automaton X after it has executed trace s. Then

$$R \in \mathrm{ref}(X, s) \iff \exists q \in \delta(X, s) \text{ s.t. } \forall \sigma \in R \;\; \delta(X, q, \sigma) = \emptyset. \tag{2.1}$$

For instance, the refusal set of the third machine after a coin is inserted is the following.

$$\{\emptyset, \{\texttt{coin}\}, \{\texttt{cookie}\}, \{\texttt{choc}\}, \{\texttt{coin}, \texttt{cookie}\}, \{\texttt{coin}, \texttt{choc}\}\}.$$

As explained in Example 2.4 the machine cannot refuse both the cookie and the chocolate bar, so the event set $\{\texttt{cookie}, \texttt{choc}\}$ is not an element of the refusal set.

In the same way as languages are used to model the behavior of deterministic systems, will the combination of languages and refusal sets be used to

model the behavior of nondeterministic systems. Automata will only be used as representation of systems and to perform computations.

Definition 2.5 A *process* is a triple $A = \big(\Sigma(A), \mathrm{L}(A), \mathrm{ref}(A)\big)$, where

$\Sigma(A) \subseteq \Sigma$ is the set of event labels,
$\mathrm{L}(A) \subseteq \Sigma(A)^*$ is the language generated by A,
for $s \in \mathrm{L}(A)$, $\mathrm{ref}(A, s) \subseteq 2^{\Sigma(A)}$ is the refusal set after s,

and which satisfies the following five conditions:

i) $\varepsilon \in \mathrm{L}(A)$,
ii) $\mathrm{L}(A) = \overline{\mathrm{L}(A)}$,
iii) $s \in \mathrm{L}(A) \;\Rightarrow\; \emptyset \in \mathrm{ref}(A, s)$,
iv) $s \in \mathrm{L}(A) \wedge R \in \mathrm{ref}(A, s) \wedge R' \subseteq R \;\Rightarrow\; R' \in \mathrm{ref}(A, s)$,
v) $s \in \mathrm{L}(A) \wedge R \in \mathrm{ref}(A, s) \;\Rightarrow\; R \cup \rho(\mathrm{L}(A), s) \in \mathrm{ref}(A, s)$.

These conditions state respectively that the language has to be nonempty and prefix closed, the refusal sets have to be nonempty and closed under the operation of taking the subset, and events that cannot be refused must be in the language [23].

For $s \notin \mathrm{L}(A)$ the refusal set $\mathrm{ref}(A, s)$ is defined to be $2^{\Sigma(A)}$. Let $\Pi(\Sigma)$ be the set of all processes A with $\Sigma(A) = \Sigma$.

It can be shown that there exists an automaton which represents a given $(\Sigma, \mathrm{L}, \mathrm{ref})$-triple if and only if the triple satisfies the conditions *i*–*v*.

Sometimes we will refer to the $(\Sigma, \mathrm{L}, \mathrm{ref})$-triple with empty language (and $\mathrm{ref}(., s) = 2^{\Sigma}$ for all $s \in \Sigma^*$) as the *empty process*, although it is officially not a process, because it violates condition *i*.

The ref-function associates to each string a set of subsets of Σ. If a subset R is an element of $\mathrm{ref}(A, s)$ then the process has the possibility after trace s to block all events in R. That is, if a user offers (via the synchronous composition defined below) to the system a set of events, which is in the refusal set, then the system has the possibility to block all these events. No event can be executed. This is called a deadlock

Definition 2.6 System A can *deadlock* after trace $s \in \mathrm{L}(A)$ if $\Sigma(A) \in \mathrm{ref}(A, s)$. System A is *deadlock-free* after s if $\Sigma(A) \notin \mathrm{ref}(A, s)$.

It will be assumed that if A is deadlock-free after trace s then eventually it will execute an event from $\lambda(\mathrm{L}(A), s)$. So a system will continue unless it deadlocks. Note however, that if a process *can* deadlock after a trace then this does not mean that it actually *will* deadlock. If a process can deadlock after trace s, then, according to 2.1, it can reach a state q_1 in which it cannot execute any further event. But it could be, because of nondeterminism, that it can also reach another state, say q_2, in which it *can* execute an event.

Deterministic Processes

In Section 2.1 automaton X is called deterministic if from each observation $s \in \mathrm{L}(X)$ it is uniquely determined in which state system X is. So it is also uniquely determined which events can be executed, and which events can be refused after s. A process will be called deterministic if any event that can be executed after a trace cannot be refused after the same trace.

Definition 2.7 Process A is called *deterministic* if for all $s \in \mathrm{L}(A)$,

$$R \in \mathrm{ref}(A, s) \iff R \subseteq \rho(\mathrm{L}(A), s).$$

The class of deterministic processes does not correspond exactly with the class of deterministic automata. Some nondeterministic automata have a deterministic process representation. Consider for instance the automaton shown in Figure 2.3. Although the automaton is nondeterministic by the definition given in Section 2.1, its behavior is clearly deterministic. After any observation it is clear which events can be executed and which will be refused.

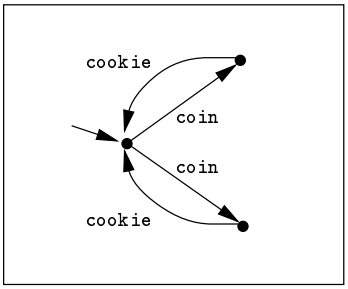

Figure 2.3: A nondeterministic vending machine?

In the sequel we will call a discrete event system deterministic if its process representation is deterministic.

It was shown in Section 2.1 that a nonempty prefix closed language $K \subseteq \Sigma^*$ defines a deterministic automaton, which is called the canonical automaton representation of language K. The following construction gives the process representation of this deterministic system.

$$\mathrm{Det}(K) = (\Sigma,\ K,\ \mathrm{ref}(K)),$$

where

$$\mathrm{ref}(K, s) \quad = \quad 2^{\rho(K,\, s)}, \quad \text{for all } s \in K.$$

Proposition 2.8 *Let $K \subseteq \Sigma^*$ be a nonempty prefix closed language. Then* $\mathrm{Det}(K) \in \Pi(\Sigma)$.

Proof. We have to prove that $\mathrm{Det}(K)$ satisfies points $i - v$ of Definition 2.5. As $\mathrm{L}(\mathrm{Det}(K)) = K$ and K is nonempty and prefix closed, it automatically follows that $\mathrm{Det}(K)$ satisfies points *i* and *ii*. Point *iii* is satisfied because $\emptyset \subseteq \rho(K, s)$ for all $s \in K$. Points *iv* and *v* follow directly from the construction of $\mathrm{ref}(\mathrm{Det}(K), s)$. □

Operations on Processes

Control will be enforced by synchronization on common events. The controlled system (i.e. the synchronous composition of the plant and the supervisor) can only execute those events that both the supervisor and the plant can execute. Below the synchronous composition of two processes with equal alphabets is defined. The synchronous composition of two processes with different alphabets is defined in Section 5.1.

Definition 2.9 Let A and B be two processes with the same alphabet. The *synchronous composition* of processes A and B, denoted $A||B$, is defined by the following relations.

$$\begin{aligned}
\Sigma(A||B) &= \Sigma(A) = \Sigma(B), \\
\mathrm{L}(A||B) &= \mathrm{L}(A) \cap \mathrm{L}(B), \\
\mathrm{ref}(A||B, s) &= \{R_\mathrm{a} \cup R_\mathrm{b} \subseteq \Sigma(A||B) : R_\mathrm{a} \in \mathrm{ref}(A, s),\ R_\mathrm{b} \in \mathrm{ref}(B, s)\}.
\end{aligned}$$

It is not difficult to show that for processes A, $B \in \Pi(\Sigma)$ the synchronous product $A||B$ is also a process. That is, $\Pi(\Sigma)$ is closed under synchronous composition.

The nondeterministic choice between processes A and B is the process $A \sqcup B$, that behaves either as process A or as process B. The selection between them is made internally inside the process $A \sqcup B$. The environment of process $A \sqcup B$ can neither observe nor influence this selection.

Definition 2.10 Let A and B be two processes with the same alphabet. The *nondeterministic choice* between A and B, denoted by $A \sqcup B$, is defined by the following equations.

$$\begin{aligned}
\Sigma(A \sqcup B) &= \Sigma(A) = \Sigma(B), \\
\mathrm{L}(A \sqcup B) &= \mathrm{L}(A) \cup \mathrm{L}(B), \\
\mathrm{ref}(A \sqcup B, s) &= \begin{cases}
\mathrm{ref}(A, s), & \text{if } s \in \mathrm{L}(A) - \mathrm{L}(B), \\
\mathrm{ref}(B, s), & \text{if } s \in \mathrm{L}(B) - \mathrm{L}(A), \\
\mathrm{ref}(A, s) \cup \mathrm{ref}(B, s), & \text{if } s \in \mathrm{L}(A) \cap \mathrm{L}(B).
\end{cases}
\end{aligned}$$

If A and B are processes then so is $A \sqcup B$. Let $\mathcal{A}$ be a possibly infinite set of processes, with all elements having the same alphabet. Let $\Sigma(\mathcal{A})$ be this alphabet. Then $\bigsqcup \mathcal{A}$ is the nondeterministic choice of all processes in A.

$$\begin{aligned}
\Sigma(\bigsqcup \mathcal{A}) &= \Sigma(\mathcal{A}), \\
\mathrm{L}(\bigsqcup \mathcal{A}) &= \bigcup\{\mathrm{L}(A) : A \in \mathcal{A}\}, \\
\mathrm{ref}(\bigsqcup \mathcal{A}, s) &= \bigcup\{\mathrm{ref}(A, s) : A \in \mathcal{A} \text{ s.t. } s \in \mathrm{L}(A)\}, \qquad \forall s \in \mathrm{L}(\bigsqcup \mathcal{A}).
\end{aligned}$$

If all elements of $\mathcal{A}$ are processes then so is $\bigsqcup \mathcal{A}$.

A process controlling process $A \sqcup B$ must be able to control both processes without knowing which process is selected.

2.3. Behavior State Representation

For processes there is no standard, well established, automaton representation such as the canonical automaton representation used for languages. In this section we will define a representation that will be used in the rest of the thesis as automaton representation of processes. It will be used to describe processes and to perform computations on processes. The representation is based on an equivalence relation similar to the Nerode equivalence relation used for languages [26].

Let A/s be the process that behaves as process A after it has executed trace $s \in \Sigma(A)$.

$$A/s = (\Sigma(A),\ \mathrm{L}(A)/s,\ \mathrm{ref}(A/s)),$$

where

$$\mathrm{ref}(A/s, v) = \mathrm{ref}(A, sv) \qquad \text{for all } v \in \Sigma(A)^*.$$

Note that if $s \notin \mathrm{L}(A)$ then A/s is the empty process.

In the same way as is done with Nerode equivalence for languages, consider traces s and s' equivalent if $A/s = A/s'$.

$$s \equiv_A s' \iff A/s = A/s'.$$

Let $[s]_A$ denote the equivalence set induced by this equivalence relation containing trace s.

$$[s]_A = \{s' \in \Sigma^* : A/s = A/s'\}.$$

One can regard $[s]_A$ as the state reached after trace s. To differentiate this notion of state from the states used in regular nondeterministic automata, we will call $[s]_A$ the *behavior state* reached after trace s.

Note that $[s]_A$ is also defined for traces $s \notin \mathrm{L}(A)$. It can be shown that if traces s and s' are not in the language of A, then $[s]_A = [s']_A$.

Lemma 2.11 *Let $s, s' \in \Sigma(A)^*$, $s, s' \notin \mathrm{L}(A)$. Then $[s]_A = [s']_A$.*

Proof. Let $s \in \Sigma(A)^* - \mathrm{L}(A)$. Then $\mathrm{L}(A/s) = \{v \in \Sigma(A)^* : sv \in \mathrm{L}(A)\} = \emptyset$, and, according to Definition 2.5, $\mathrm{ref}(A/s, v) = 2^{\Sigma(A)}$ for all $v \in \Sigma(A)^*$. So, for all $s' \in \Sigma(A)^* - \mathrm{L}(A)$, $\mathrm{L}(A/s) = \emptyset = \mathrm{L}(A/s')$, and for all $v \in \Sigma(A)^*$, $\mathrm{ref}(A/s, v) = 2^{\Sigma(A)} = \mathrm{ref}(A/s', v)$. Hence $[s]_A = [s']_A$. □

Let the *dump state* be the equivalence set containing all traces that are not in the language of A.

Lemma 2.12 *Let $s, s' \in \Sigma(A)^*$ such that $[s]_A = [s']_A$. Then*

$$s \in \mathrm{L}(A) \iff s' \in \mathrm{L}(A)$$

Proof. If $s \in \mathrm{L}(A)$ then $\varepsilon \in \mathrm{L}(A/s)$. So, by $[s]_A = [s']_A$, $\varepsilon \in \mathrm{L}(A/s')$. This implies that $s' \in \mathrm{L}(A)$. The converse holds by symmetry. □

Definition 2.13 The *behavior state representation* of process A is a basic deterministic automaton extended with an extra element. This extra element is denoted $\mathrm{ref}(A)$. It maps behavior states to the corresponding refusal sets. The behavior state representation of process A is defined by

$$\big(\Sigma(A),\, Q(A),\, \delta(A),\, Q_0(A),\, \mathrm{ref}(A)\big),$$

where

$$\begin{aligned} Q(A) &= \{[s]_A : s \in \mathrm{L}(A)\}, \\ Q_0(A) &= [\varepsilon]_A, \end{aligned}$$

and for all $[s]_A \in Q(A)$

$$\begin{aligned} \delta(A, [s]_A, \sigma) &= \begin{cases} [s\sigma]_A, & \text{if } \sigma \in \lambda(\mathrm{L}(A), s), \\ \emptyset & \text{otherwise,} \end{cases} \\ \mathrm{ref}(A, [s]_A) &= \mathrm{ref}(A, s). \end{aligned}$$

Note that $Q(A)$ does not contain the dump state. Occasionally, also the behavior state space including the dump state will be used. Define

$$Q^+(A) = \{[s]_A : s \in \Sigma^*\}.$$

Although process A may be nondeterministic, the behavior state representation is always a deterministic automaton. It can be seen as if the nondeterministic properties of process A are encoded inside the refusal sets of the behavior states instead of modeled by the transition function.

In Figure 2.4 the behavior state representation of the process shown in Figure 2.2.c is given. For compactness reasons, only the maximal refusals, i.e. the refusals not strictly contained in another refusal, are shown. As refusal sets are closed under the operation of taking subsets, the whole refusal set can

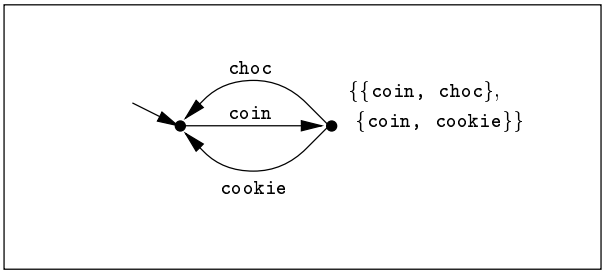

Figure 2.4: Behavior state representation of a vending machine.

be derived from the maximal refusals. Also the refusal sets of states $[s]_A$ with $\mathrm{ref}(A, [s]_A) = 2^{\rho(\mathrm{L}(A),s)}$ are not shown. These refusals can be derived from the outgoing arrows of the state.

The refusal set of state $[\varepsilon]_A$ is

$$\mathrm{ref}(A, [\varepsilon]_A) \;=\; 2^{\rho(\mathrm{L}(A),\varepsilon)} \;=\; \{\emptyset,\, \{\mathtt{choc}\},\, \{\mathtt{cookie}\},\, \{\mathtt{choc}, \mathtt{cookie}\}\}.$$

The refusal set of state $[\mathtt{coin}]_A$ is the set of all subsets of $\{\mathtt{coin}, \mathtt{cookie}\}$ and $\{\mathtt{coin}, \mathtt{choc}\}$. It is shown in Section 2.2.

Converting a nondeterministic finite state machine to a behavior state representation is basically the same as converting a nondeterministic state machine to a deterministic version. This conversion has a known complexity that is worst case exponential in the size of the state space of the original state machine. But in practice systems have sufficient structure, such that this conversion may not be a problem.

2.4. Specification, Implementation, and Control

In general a design problem can be defined as: given a specification, find an implementation that satisfies the specification. A design problem can be considered a supervisory control problem if the implementation consists of an already existing uncontrolled process G and a still to be designed supervisor process S. In this section we will define the control problem of finding a supervisor S such that $G||S$ can replace a given specification process E. The following example will illustrate in what sense an implementation must be able to replace a specification.

Example 2.14 A system usually does not work on its own. It is embedded in a larger system. For instance a hard-disk unit is used inside a computer system. The computer is usually designed in a different place than the hard-disk unit. During the design phase a standard is negotiated between the computer manufacturer and the disk manufacturer. This standard is the specification of the hard-disk. After this standard is established the computer designer models

a computer system in which it expects a hard-disk unit that behaves according to this specification. It is the hard-disk developers task to build a hard-disk unit that satisfies this specification. Without him knowing how the computer system will look, he has to design a unit that works together with this system.

Consider the following implementation relation [15, 23] .

Definition 2.15 Let $A, B \in \Pi(\Sigma)$. *A reduces B*, denoted by $A \sqsubseteq\!\!\!\!\sim\ B$, if

i) $\mathrm{L}(A) \subseteq \mathrm{L}(B)$, and

ii) $\mathrm{ref}(A, s) \subseteq \mathrm{ref}(B, s)$ for all $s \in \mathrm{L}(A)$.

Here, point *i* states that system A may only do what system B allows, and point *ii* states that A may only refuse what B can also refuse. We will say that process $G||S$ implements specification E if $G||S \sqsubseteq\!\!\!\!\sim\ E$.

Note that $A = B$ if and only if $A \sqsubseteq\!\!\!\!\sim\ B$ and $B \sqsubseteq\!\!\!\!\sim\ A$. This property will be used in many proofs. The next result is well known from computer science [10]. It states that the reduction relation forms a congruence with the synchronous composition.

Proposition 2.16 *Let $A_1, A_2, B_1, B_2 \in \Pi(\Sigma)$ such that $A_1 \sqsubseteq\!\!\!\!\sim\ A_2$ and $B_1 \sqsubseteq\!\!\!\!\sim\ B_2$. Then $A_1||B_1 \sqsubseteq\!\!\!\!\sim\ A_2||B_2$.*

In Example 2.14 the implementation of the hard-disk has to be such that it can replace its specification in any computer system. This is guaranteed by the reduction relation. Let $G||S$ stand for the implementation of the hard-disk, E for the specification, and C for the rest of the computer system. Then the following implication, which is a direct consequence of Proposition 2.16, states that $G||S$ can replace E in any computer system.

$$G||S \sqsubseteq\!\!\!\!\sim\ E \quad \Rightarrow \quad \forall C,\ (G||S)||C \sqsubseteq\!\!\!\!\sim\ E||C. \tag{2.2}$$

This implication shows that the reduction relation is strong enough to use it as an implementation relation. The following result shows that it also forms a necessary condition to guarantee deadlock-free behavior. This result forms the main motivation for the use of failure semantics and the reduction relation.

Theorem 2.17 *Let $A, E \in \Pi(\Sigma)$.*

$$\begin{array}{ll} & A \sqsubseteq\!\!\!\!\sim\ E \\ & \Longleftrightarrow \\ \forall C \in \Pi(\Sigma) & \mathrm{L}(A||C) \subseteq \mathrm{L}(E||C), \quad \textit{and} \\ & E||C \textit{ deadlock-free } \Rightarrow\ A||C \textit{ deadlock-free.} \end{array}$$

Proof. The $\Rightarrow$-part follows from Proposition 2.16. For the proof of the $\Leftarrow$-part assume that A does not reduce E. Then either $\mathrm{L}(A) \not\subseteq \mathrm{L}(E)$ or there exists an $s \in \mathrm{L}(A)$ such that $\mathrm{ref}(A,s) \not\subseteq \mathrm{ref}(E,s)$. Assume there exists an $s \in \mathrm{L}(A)$ such that $s \notin \mathrm{L}(E)$. Let C be a process such that $s \in \mathrm{L}(C)$. Then $s \in \mathrm{L}(A||C)$ but $s \notin \mathrm{L}(E||C)$, so $\mathrm{L}(A||C) \not\subseteq \mathrm{L}(E||C)$. For the other alternative let $s \in \mathrm{L}(A)$ such that there exists an $R \in \mathrm{ref}(A,s)$ and $R \notin \mathrm{ref}(E,s)$. Let C be a process such that $\mathrm{ref}(C,s) = 2^{\Sigma - R}$. Then $\Sigma = R \cup (\Sigma - R) \in \mathrm{ref}(A||C,s)$, but $\Sigma \notin \mathrm{ref}(E||C,s)$. So $E||C$ is deadlock-free, but $A||C$ is not. □

The basic supervisory control problem can be formulated as follows. Given an uncontrolled system G and a specification E, find a supervisor S such that $G||S \sqsubseteq_{\approx} E$.

In some applications the supervisor does not have the ability to block all events. For instance if an alarm event is executed when some water level exceeds a threshold, then this event can be observed by the supervisor but it cannot be blocked. If this event has to be prevented from occurring then somewhere else in the system some other events have to be blocked (for instance the event corresponding with the closing of a waste gate) such that the alarm event cannot be executed.

Usually the presence of uncontrollable events is modeled by splitting up the event set Σ into controllable and uncontrollable events, Σ_{c} and Σ_{uc} respectively. A supervisor is called complete if it does not block any uncontrollable events.

Definition 2.18 Supervisor S is *complete* (w.r.t process G) if

$$\forall s \in \mathrm{L}(G||S),\ \forall R_{\mathrm{s}} \in \mathrm{ref}(S,s),\ R_{\mathrm{s}} \cap \Sigma_{\mathrm{uc}} \subseteq \rho(\mathrm{L}(G),s).$$

Definition 2.19 Let the uncontrolled system $G \in \Pi(\Sigma)$ and a specification $E \in \Pi(\Sigma)$ be given. The *basic supervisory control problem* is to find a complete supervisor $S \in \Pi(\Sigma)$, such that $G||S \sqsubseteq_{\approx} E$.

2.5. Comparison of Frameworks

In this section we will compare the approach based on failure semantics and the reduction relation with other approaches. The comparison is not intended to be complete. It just illustrates some differences between the approaches.

The original framework introduced by Ramadge and Wonham [52] was intended to handle only deterministic systems. The framework presented in this thesis is also capable of handling nondeterministic systems. But even if we restrict our attention to deterministic systems there are some important differences.

It can be shown in the framework presented by Ramadge and Wonham that the corresponding implication of (2.2) is not satisfied. In that framework a discrete event system, A, is modeled by the triple $(\Sigma(A), \mathrm{L}(A), \mathrm{L_m}(A))$, where

$\mathrm{L}(A) \subseteq \Sigma(A)^*$ is the prefix closed language that A can generate and $\mathrm{L_m}(A) \subseteq \mathrm{L}(A)$ is the language that A accepts or marks.

Definition 2.20 Let A and B be discrete event systems. $A \sqsubseteq_\mathrm{m} B$ if

i) $\mathrm{L}(A) \subseteq \mathrm{L}(B)$,
ii) $\mathrm{L_m}(A) \subseteq \mathrm{L_m}(B)$,
iii) $\mathrm{L}(A) = \overline{\mathrm{L_m}(A)}$.

A system is called *M-nonblocking* if it satisfies point *iii*.

System $G||S$ is considered an implementation of E in the Ramadge Wonham framework if $G||S \sqsubseteq_\mathrm{m} E$. Note that if $\mathrm{L}(E) = \overline{\mathrm{L_m}(E)}$ then point *ii* and *iii* together imply point *i*. Usually the specification is not given as a process but as a language $K \subseteq \mathrm{L_m}(G)$. In this case the specification process E can be defined by $\mathrm{L}(E) = \overline{K}$ and $\mathrm{L_m}(E) = K$. Sometimes a non-marking supervisor is required, that is $\mathrm{L_m}(S) = \mathrm{L}(S)$. In this case it is usually assumed that $\mathrm{L_m}(E) = \mathrm{L}(E) \cap \mathrm{L_m}(G)$. These differences are not important for the following discussion, which mainly concerns point *iii*.

We want that an implementation can replace the specification in any environment. This is however not guaranteed by the $\sqsubseteq_\mathrm{m}$ relation. The next example illustrates that in general it cannot be guaranteed that the implementation is M-nonblocking in any environment, i.e.

$$G||S \sqsubseteq_\mathrm{m} E \wedge E||C \text{ is M-nonblocking } \not\Rightarrow \ (G||S)||C \text{ is M-nonblocking.}$$

Example 2.21 Let E be the specification with $\mathrm{L_m}(E) = \mathtt{a(b+c)}$ and $\mathrm{L}(E) = \overline{\mathtt{a(b+c)}}$. Let $A = G||S$ be the implementation with $\mathrm{L_m}(A) = \mathtt{ab}$ and $\mathrm{L}(A) = \overline{\mathtt{ab}}$. And let C represent the rest of the computer system with $\mathrm{L_m}(C) = \mathtt{ac}$ and $\mathrm{L}(C) = \overline{\mathtt{ac}}$. Observe that $A \sqsubseteq_\mathrm{m} E$, but $\mathrm{L}(A||C) = \mathtt{a}$ and $\mathrm{L_m}(A||C) = \varepsilon$, thus $\mathrm{L}(A||C) \neq \overline{\mathrm{L_m}(A||C)}$. So $A||C \not\sqsubseteq_\mathrm{m} E||C$.

It can be derived from results obtained by Wonham and Ramadge [63] on modular control that $(G||S)||C$ is only M-nonblocking if $\mathrm{L_m}(G||S)$ and $\mathrm{L_m}(C)$ are *non-conflicting*. That is, processes A and B are non-conflicting if common prefixes in both processes can be extended to a common marked trace.

$$\overline{\mathrm{L_m}(A) \cap \mathrm{L_m}(B)} = \overline{\mathrm{L_m}(A)} \cap \overline{\mathrm{L_m}(B)}.$$

This constraint also limits the use of modular control in the Ramadge Wonham framework. If a specification E can be decomposed as $E_1||E_2 = E$, then it has computational advantages to first synthesize both S_1 and S_2 such that $G||S_1$ implements E_1 and $G||S_2$ implements E_2. In the framework based on failure semantics it can be deduced from Proposition 2.16 and the fact that $G \sqsubsetneq G||G$, that

$$G||S_1 \sqsubseteq_{\approx} E_1 \ \wedge \ G||S_2 \sqsubseteq_{\approx} E_2 \ \Rightarrow \ G||S_1||S_2 \sqsubseteq_{\approx} E_1||E_2.$$

In the Ramadge Wonham framework however it is necessary that $\mathrm{L_m}(G||S_1)$ and $\mathrm{L_m}(G||S_2)$ are non-conflicting in order to guarantee that $G||S_1||S_2$ is M-nonblocking. This constraint is often not easy to satisfy.

The discussion above considers how well the M-nonblocking property and the deadlock freeness property behave within their own framework. It does not compare the properties directly with each other. The M-nonblocking property states that a process is always able to complete a task, whereas the deadlock freeness property states that a process is always able to continue. Note that it cannot be specified by a marked language that the implementation should be deadlock free. Even if there are transitions leading out of each marked state in the specification, then still an implementation which deadlocks in a marked state satisfies the specification according to Definition 2.20. So marking cannot be used to guarantee deadlock free behavior. It depends on the particular application which approach is more suited.

The marking condition on states can be replaced by an event that indicates the completion of a task. This issue is treated in Section 4.6. With this approach a process can be considered nonblocking if it cannot refuse such a task completion event. The nonblocking property can then be adequately handled within the framework based on failure semantics.

Within the computer science area synthesis is investigated based on infinite trace theory [18, 38, 48]. Also within the control theory area this approach has been followed [58]. Infinite trace automata have an acceptance condition, which is similar to the marking condition for finite trace automata. Because of this acceptance condition the corresponding implication of (2.2) will not be satisfied within this framework. Also, there will be extra constraints necessary for modular control synthesis.

Nondeterministic Specifications

It is logical, if one considers that the implementation should be able to replace the specification, that the specification is given as a process. In the rest of this section it will be shown that this specification method has more expressive power than a specification given as a language or as a range of languages.

Initially in discrete event control the problem was posed to find a supervisor S such that

$$\mathrm{L}(G||S) \quad = \quad \mathrm{L}(E).$$

Conditions were found under which such a solution exists. But this control problem formulation is rather rigid. It does not allow for any flexibility. Therefore the specification was considered to denote the set of all legal traces and the following more flexible control problem was posed. Find a supervisor S such that

$$\mathrm{L}(G||S) \quad \subseteq \quad \mathrm{L}(E).$$

The task was to find the largest solution of this control problem. However, this control problem formulation allowed to many implementations. For instance the language $\{\varepsilon\}$ also satisfies the inclusion relation. Two solutions were presented to restrict the class of legal implementation. The one is marking, which we discussed above and will not be considered here. The other is the minimal allowable language, usually denoted $\mathrm{L}(A)$. The control problem with the minimal allowable language is to find a supervisor S such that

$$\mathrm{L}(A) \quad \subseteq \quad \mathrm{L}(G||S) \quad \subseteq \quad \mathrm{L}(E)$$

The minimal allowable language is also used to synthesize a solution if the supremal solution cannot be synthesized. However, it is not always possible to define an adequate minimal allowable language.

Example 2.22 Consider a car. If one wants to describe how a car should be operated, one should include cars with a manually operated gearbox, and cars with an automatic transmission. As the intersection of the language needed to describe the automatic transmission, and the language needed to describe the manual gearbox is empty, the minimal allowable language will be empty as well. But cars do need a mechanism to drive their wheels. Thus, a minimal allowable language cannot describe all minimal allowable behavior. If one wants to specify a minimal allowable language, then one already has to make the choice whether the car will have a manual gearbox or an automatic transmission. So, one already has to decide on some design issues in the minimal allowable language. In a control context this would mean that the minimal allowable language already specifies some parts of the solution to the control problem.

Using a nondeterministic specification, both options can be included by allowing the nondeterministic choice between the automatic transmission and the manual gearbox. The specification process can be seen as a representation of the set of all legal implementations.